Enhancing MVC architecture pattern description using its System of Systems model

Mrityunjay Kumar
IIIT Hyderabad
India
mrityunjay.k@research.iiit.ac.in

Venkatesh Choppella
IIIT Hyderabad
India
venkatesh.choppella@iiit.ac.in

ABSTRACT

When novice engineers (fresh or recent graduates with little industry experience) join a SaaS (Software-as-a-Service) product company, they are tasked with comprehending the product, especially its behavior and dynamics. We believe that they can comprehend more effectively if they know and understand the architecture patterns used in the product.

Are the current architecture pattern descriptions of high quality? Do they fit the needs of novice engineers?

We evaluated the pattern descriptions of Model-View-Controller (MVC) (a popular and important architecture pattern for cloud systems) from a quality and fitment perspective and found gaps. To address these gaps, we have built a System of Systems (SoS) model of MVC that uses a transition systems vocabulary and a set-theoretic notation. In the paper, we show that this SoS model provides a rich set of information about the behavior and dynamics of the MVC components and their interactions. The model bridges the gaps in the MVC pattern description.

One of the contributions of the paper is to provide criteria to evaluate the pattern descriptions for quality and fitment for novice engineers. The paper proposes that we augment the benchmark pattern description of MVC with an SoS model. The paper also demonstrates a general approach to building SoS models for architecture patterns and recommends creating a catalog of SoS models for SaaS architecture patterns. We believe such a catalog will significantly help novice engineers in comprehension and other software engineering activities.

CCS CONCEPTS

• **Software and its engineering** → *Software architectures*; **Software system models**.

KEYWORDS

architecture pattern, Model-View-Controller (MVC), novice engineers, transition systems, system of systems

ACM Reference Format:
Mrityunjay Kumar and Venkatesh Choppella. 2024. Enhancing MVC architecture pattern description using its System of Systems model. In *17th Innovations in Software Engineering Conference (ISEC 2024), February 22–24, 2024, Bangalore, India.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3641399.3641410

1 INTRODUCTION

Every new hire in a SaaS (Software-as-a-Service) product company must understand the product well before contributing to the code base. Given a comprehension task and access to the product code repository, a typical approach by the engineer is to read this large code base and attempt to understand it. Prior programming experience and the domain knowledge acquired through various overview sessions during onboarding are leveraged to do so. Sometimes, partial product documentation may be available and can be somewhat useful. This approach usually proves ineffective since it is hard for a novice to understand a complex system by focusing on such granular details as code [22, 23]. A top-down [9, 10], knowledge-based [30], systems way [21] is likely to be a better approach.

We believe that understanding the architecture patterns used in the product can help novice engineers comprehend them more effectively.

Architecture patterns provide established solutions to architectural problems. They offer a common vocabulary and understanding of the underlying principles and capture decisions made frequently in practice. Knowledge of the principles and decisions can provide important cues to understand the behavior of a running system. The vocabulary can facilitate efficient communication and knowledge transfer. It can also help capture the product understanding in a precise way.

The knowledge of patterns can aid comprehension of a software system in three ways: a) The engineer can look for the signature of a pattern in the product (the components, the connections, and interactions between components), which decomposes the comprehension tasks into smaller, more manageable activities, b) the dynamics described in the pattern can help make sense of specific parts of the code which implement the dynamics, and c) The knowledge of the design decisions of a pattern can explain why specific components or algorithms have been implemented in a non-obvious way.

However, for these pattern descriptions to benefit novice engineers, they must be high-quality and fit their needs. High-quality pattern descriptions are clear comprehensive, and articulate the architectural solution precisely. Fitment comes from the right level of abstraction and a vocabulary that is easy to learn and use. High-quality and fit pattern descriptions can be a reference and aid in teaching-learning.

We want to explore whether the current architecture pattern descriptions are high-quality and fit novice engineers' needs. In this paper, we report on a study we did of the pattern description of a popular and important architecture pattern for SaaS products (MVC pattern). We framed three research questions to guide our study:

RQ1: Which source of documentation of the MVC pattern description can be considered a benchmark?
RQ2: Is the benchmark high-quality and fit for the needs of novice engineers?
RQ3: How can we extend the benchmark to improve its quality and fitness?

In the paper, we show that the benchmark description of the MVC pattern has gaps and recommend a novel way of augmenting the descriptions to address these gaps. We propose that we build a System of Systems (SoS[1]) model of MVC and use the model representation as an augmented description of the patterns. We show that a pattern can be modeled as a System of Systems (where the components are systems) and that transition systems vocabulary can be used to capture the system dynamics in a precise way.

Section 2 presents the related work on what constitutes good documentation of pattern description and why systems vocabulary can produce a good pattern description. Using the literature, we also create the evaluation criteria for a pattern description for its quality and fitment. Section 5 presents the transition systems model for communicating systems, and Section 6 shows an example of an SoS model. Section 3 describes the MVC pattern as presented in the literature, and Section 7 builds the SoS model of MVC, using the vocabulary presented in Section 5. Section 8 includes a short survey of MVC pattern descriptions available through various sources. The section also presents an evaluation of the benchmark pattern description [11] using the criteria from Section 2.1 and shows how the SoS model description addresses the gaps. Section 9 concludes the paper and outlines the future work planned.

2 RELATED WORK

Why architecture patterns: Perry and Wolf [38] suggested modeling an architecture as a three-tuple {Elements, Form, Rationale}, and defined architectural style (or pattern) as an abstraction that is less constrained and less complete than a particular architecture. Bass et al. define an architecture pattern as a package of design decisions found repeatedly in practice, has known properties that permit reuse, and describes a class of architectures [6]. Patterns are helpful because they provide a common vocabulary and understanding of the underlying principles [18].

Documenting architecture patterns and their behavior well: When documenting patterns, Clements et al. [14] suggest capturing assumptions about the environment, as well as assumptions about the need. The documentation should also capture the constraints imposed to shape the architecture in specific ways. Shaw and Garlan [39] talk about documenting the decisions made when developing the architecture.

Architecture patterns are documented by specifying the context, problem, and solution [11]. A *context* represents a recurring, common situation that gives rise to a *problem*, which is solved by the approach provided in the *solution*. The solution should contain "*a set of element types, a set of interaction mechanisms, a topological layout of components, and a set of semantic constraints on topology, element behavior, and interaction mechanisms*"[6].

[1]SoS is used here differently from how it is used elsewhere: "systems that are built from components which are large-scale systems in their own right."

Documentation of behavior should reveal information about the ordering of interactions among the elements and time dependencies of interactions so that one can reason about the system's completeness, correctness, and quality attributes [5]. There have been attempts to document the architecture and patterns more formally using Architecture Description Languages (ADL) [4, 26], and there have been many ADLs defined and used over the years [34]. However, they have focused more on the structure and quality attributes and less on the behavior.

SaaS products leverage common architectural patterns: Service-oriented architectures, for example, microservices-based architectures, are common patterns for SaaS products [43]. While there is a case for specific architecture patterns for cloud applications [37], many existing architecture patterns are used in cloud application development as well [16]. Hence, knowing existing, common architecture patterns well can benefit novice engineers at SaaS product companies.

An architecture pattern is a system: A system is "*a set of elements that is coherently organized and interconnected in a pattern or structure that produces a characteristic set of behaviors*" [35]. This definition is aligned with how we define an architecture or architecture pattern [6, 11] and suggests the use of systems modeling for architecture patterns. When we model an architecture or an architecture pattern as a system, we can apply the principles and vocabulary that Systems theory [2] affords us.

There are multiple ways to model and describe a system - we can use DEVS formalism [47], Discrete-time dynamical systems formalism [7], Abstract State Machine formalism [8, 19], I/O automata [31, 32], SysML [25, 45] or Statecharts [20], to name a few.

Process calculi are good candidates for describing the behavior of a system as well. Milner's CCS [36] has been used for modeling MVC [46], pi-calculus [36] extensions being suitable for communicating systems that we are interested in. Hoare's CSP [24] serves a similar purpose. However, for the kind of interactive systems, we are interested in, process calculi are better used when paired with a specification language like Z [40, 41], which can specify non-behavioral aspects. In contrast, CCS or CSP can be used for behavior description [33].

Given our focus on cloud-based reactive systems and target audience of novice engineers, we plan to use labeled transition systems [44] as used in [13] and adapt it for our modeling of communicating systems and SoS by using the notion of system composition by Tabuada [42]. This enables us to have a small vocabulary and a single set of notation, using a simple state machine-based computation model that can serve novice engineers' needs well. We have used a similar notation in prior work on the design skills of novice engineers [29] and also evaluated it for ease of learning and use [28].

2.1 Evaluation criteria for pattern description

Based on the above, we can identify the attributes of a **high-quality and fit pattern description**.

2.1.1 High-quality. The description should provide clear and precise information about the following:

(1) Q1: Elements of the pattern
(2) Q2: Connections among the elements
(3) Q3: Behavior of the individual elements
(4) Q4: Behavior of the overall system (interaction among elements, including timing dependencies of these interactions, requirements, and constraints on the interactions)

2.1.2 Fit for novice engineers. The description should be clear and understandable in the following ways:

(1) F1: Level of abstraction. Information should be available at a level of abstraction higher than code (so that implementation details don't become distracting).
(2) F2: Modes of documentation. It should use modes of documentation that afford precision - mathematical notations, formal pseudo-code, and formal languages.
(3) F3: Vocabulary. It should have a small vocabulary and be easy to learn and use.

3 ARCHITECTURE PATTERN: MODEL-VIEW-CONTROLLER(MVC)

MVC (Model-View-Controller) is a popular architecture pattern used over the years to architect user interfaces for desktop, mobile, and web-based applications (along with its many variations that have evolved over the years). We will use the case study of the MVC pattern to illustrate the modeling as SoS. This section describes the pattern and how they are described in the literature.

3.1 Pattern overview

This section is adapted from two books: "Pattern-oriented Software Architecture (POSA)" by Buschmann et al. [11] and "Software Architecture in Practice" by Bass et al. [6] and is meant to give an overview of the pattern that we will model. Note that there are many flavors of MVC in practice; we have picked the initial formulation of the pattern.

3.1.1 Context and Problem. For interactive systems, there are many situations when one or more of these need to be addressed: a) **Interface diversity**: Different users wish to have different user interfaces to interact with the same data, b) **Change velocity**: Changes in user interfaces are rapid and changing the core of the application at that pace is not sustainable, and c) **Data safety**: Complex rules govern the sanity and consistency of data, and it is essential to keep the data safe (consistent).

These are especially acute problems for data-driven applications.

3.1.2 Solution approach. The solution is to separate the core data and functionality from the presentation of the data (which governs the user interface) and centralize the handling of user actions so that data consistency and integrity can be maintained.

3.1.3 Solution summary. Fig 1 shows the classical MVC pattern [6]. The pattern uses three components with specific responsibilities and interaction patterns. We assume that the user interacts with the View.

(1) View is responsible for rendering the UI for the user by fetching data from the Model and enabling the user to interact with the Controller. It also receives update notifications from the Model and updates the View if required.
(2) Controller provides a safe way to manipulate the Model. It receives actions from View and processes them. Service requests arising from the actions are sent to the Model, and View updates are sent to the View.
(3) Model keeps and manages the data. It receives read requests from View and service requests from the Controller. When there are updates to the data, it lets the View know of the change.

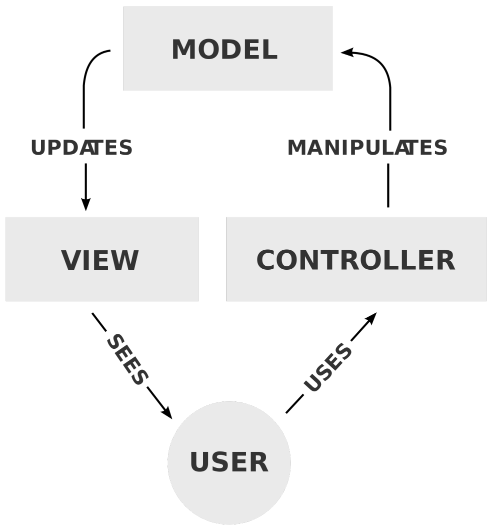

Figure 1: Diagram of interactions within one possible take on the MVC pattern (Credit: Wikipedia)

Controller is responsible for **Data safety** by filtering the user requests and only sending valid update requests to the Model. By having View own the user experience, multiple Views (and corresponding controllers) can be created for different interfaces for the same data (in Model), thus enabling **Interface diversity**. By separating the logic across three components, **Change velocity** can be sustained because changes get localized most of the time.

There are two key use cases that the user expects this pattern to handle in the same way a monolith UI application handles it. We illustrate this by using an example of a UI application that shows the user's address; depending on the postal code in the address, they are shown e-waste collection centers in that area. The two use cases are: a) UC1: Jia scrolls down on the page showing the collection centers in her area, and she sees additional collection centers on the page, and b) UC2: Ali changes his profile and adds his office address. The page shows the collection centers near his office.

4 MVC PATTERN DESCRIPTION BENCHMARK

To understand how MVC is being documented in practice, we reviewed its description from three types of sources: books [6, 11, 17], the MVC paper for Smalltalk-80 [27], and industry sources (Microsoft ASP.NET MVC [3], Oracle MVC [1])

Most of the other sources had similar content to these sources. Surprisingly, we could not find an 'official' or 'standard' MVC pattern description. Table 1 summarizes the review of these sources under three categories: Structure, Behavior/Dynamics, and Modes of Documentation.

5 MODELING COMMUNICATING SYSTEMS

Given that we are interested in systems that communicate with each other, we make these assumptions for our model: a) Systems communicate with each other by sending and receiving messages, b) Systems have ports. A port is an abstract entity through which messages are sent or received. A port is either an input or output port, but not both. It supports only one message type drawn from a universe of message types, and c) Environment supports two primitive methods to aid communication: *send()* will send a message via the given output port, and *receive()* receives a message (possibly empty if there are no messages) from the given input port.

A communicating system S is a tuple $\{X, X^0, P, f\}$ where X is the set of states, X^0 is the set of initial states, P is the set of ports, $P \subseteq Ports$, and f is the transition function, $f \subseteq (X \times MessageTypes) \times X$.

$MessageTypes$ is the universe of message types, all possible message types used by the system's ports. $Systems$ is the universe of systems. $Dirs \equiv \{in, out\}$ is an enumeration of possible types of direction associated with a port - in or out.

$Ports \subseteq Systems \times Dirs \times MessageTypes$. A port p is modeled as a tuple (s, dir, m), $s \in Systems$, $dir \in Dirs$, $m \in MessageTypes$. The port belongs to the system s, its direction of communication is dir, and it supports message type m. Without losing generality, we assume that only one message type is transmitted or received on a port (if a set of message types needs to be supported, a new message type can be declared as the union of these types).

$x' = f(x, m)$ where x' is the next state, f is the transition function, x is the current state, and m is the currently received message. The transition function f can be written as guarded commands tuple $(< guard >, < command >)$ [15]. The command is executed if the guard is true in a particular invocation of f. Only one guard should be true in any particular invocation, but if more than one guard is true, any command may be executed. For convenience and brevity, we represent these tuples in the form *if <guard> then <command>* as the body of the function $f(x, m)$.

5.1 Modeling communication channels

Communication channels can be modeled as IO automata [32], or it can be done as a transition system [13]. We define the channel and then show a transition system model.

5.1.1 ***Channel:*** A channel connects two ports, $C \subseteq Ports \times Ports$. It is a connection between two compatible ports to achieve unidirectional communication. A channel $c = (p_{out}, p_{in})$ is valid iff these conditions hold: a) $dir_{out} = out$ and $dir_{in} = in$, where dir is the direction of the port, b) $m_{out} = m_{in}$ where m is the message type of the port, and p_{out} or p_{in} are not part of any other channel.

We model the message communication on a channel like a mailbox (with message queues) and model a channel c as a system.
$S_c = \{X_c, X_c^0, U_c, f_c\}$ is the system modeling a channel where

X_c is a tuple (p_{out}, p_{in}, mq), mq is the queue to hold the message at input port p_{in}.
X_c^0 is $(p_{out}, p_{in}, < empty >)$, queue is empty at initial state.
U_c is the set of actions supported by the channel, $\{send, receive\}$
f_c is the transition function - a $send$ action adds the message to the tail of the queue, and a $receive$ action removes the message from the head of the queue.

However, to keep the overall model simple and focused on the component systems (instead of channels), we make two assumptions: a) The environment makes available two methods to all systems: $send()$ and $receive()$ and b) A message sent on an output port of a channel will be immediately available for receiving at the corresponding input port.

Given these assumptions, we can model a channel as a wire rather than a system. We use this channel model when defining an SoS.

For notational convenience, we also represent a channel c as $\{s_{out}, s_{in}, m\}$ where s is the system the port belongs to and m is the message type of the ports.

5.2 System of Systems (SoS)

When a set of communicating systems G is configured with a set of channels L (systems of G are interconnected using the channels of L), they produce a behavior distinct from the behavior of individual systems. We call this configuration a System of Systems (SoS).

An SoS is designed with an expected behavior, which can be captured in terms of specified behaviors and requirements, including any constraints. These drive the choice of individual systems, and the channels and representation of an SoS should also include this information.

In this section, we will show how SoS is formed via the composition of component systems. We will also introduce a simpler way to represent an SoS.

Given two systems, $S_1 = \{X_1, X_1^0, P_1, f_1\}$, $S_2 = \{X_2, X_2^0, P_2, f_2\}$ and a set of channels L, we define a composed system
$S_C = \{X_C, X_C^0, P_C, f_C\}$ where
$X_C = \{X_1, X_2, L\}$,
$X_C^0 = \{X_1^0, X_2^0, L\}$,
$P_C = (P_1 \cup P_2) - P^L$ where P^L is the set of all ports that are part of a channel in L.
$f_C = f_1 || f_2$ where $||$ represents parallel execution

In other words, state space of the composed system is the composition of the states of the individual systems and the set of channels, ports of the composed system are the aggregate of the ports from individual systems that are not part of any channel, and the transition function of the composed system is the parallel execution of transition functions of the component systems. This notion of composition can be extended to finitely many systems.

Given the above definition for the composition of communicating systems, we can represent an SoS in a simpler way in terms of the involved systems and channels.

An SoS is a tuple (G, L, B) where G is the set of component systems, L is the set of channels, and B is the set of expected behaviors.

5.2.1 *Representing set of systems (G).* This is the list of component systems that are included in this SoS. Each system is defined as its

Table 1: MVC Pattern documentation in literature

Documentation source	*Structure*	*Behavior/Dynamics*	*Modes of documentation*
Pattern-oriented Software architecture [12]	Prose to describe each of the components (View, Model, and Controller), Class diagram for each component, C++ implementation object diagram	Two sequence diagrams - one for the scenario where user input changes the model, another for initialization of the components	Lots of implementation details provided in C++, UML sequence diagrams used, a lot of prose
Software Architecture in Practice [6]	Prose to briefly describe each component, summarized in the form of a table	A diagram that illustrates the connection among the components and shows actions by the user and by the components	Very brief description overall, primarily prose (with table) and a diagram
Patterns of enterprise software architecture [17]	Prose to explain the rationale of MVC, diagram to describe the components and connections	Prose describes basic behavior	Very brief, mostly prose, defers to POSA for details
Smalltalk-80 [27]	Prose to explain the pattern and its components, a detailed diagram that describes the structure as well as some aspects of behavior	Prose describes basic behavior, labels on the diagram add more details	Prose, implementation details using Smalltalk objects
Microsoft ASP.NET MVC [3]	Prose to explain the pattern, a diagram for the components	Prose describes basic behavior	C# code is used to explain much of the pattern
Sun/Oracle MVC [1]	Prose to describe the pattern, a diagram (same as in Bass's book) is used as well for some aspects of structure	A diagram that describes the connection among the components and shows actions by the user and by the components	Prose, diagram, and Implementation details using Java

tuple $\{X, X^0, P, f\}$. For example, if the SoS consists of three component systems S_1, S_2 and S_3, then a set of systems $G = \{S_1, S_2, S_3\}$ where $\forall i \in \{1, 2, 3\}, S_i = \{X_i, X^0{}_i, P_i, f_i\}$.

5.2.2 Representing set of channels (L). We will represent the channel c in both ways: (s^{out}, s^{in}, m) as well as (p^{out}, p^{in}). We will enumerate the channels that belong to L.

5.2.3 Representing expected behaviors (B). We use the notion of Traces and Specifications from Hoare's CSP formalism [24] to capture the expected behavior. We capture the specified behavior using Traces and use Specifications for capturing requirements and constraints. Behavior $B = T \cup R$ where T is the set of valid traces of the SoS and R is a set of specifications captured as conditions on these traces.

Example Trace: $< read_{C1}, readresp_{C2} >$ is an example trace - it represents a trace where message 'read' on channel C1 was observed, followed by a message 'readresp' on channel C2.

Example Specification: $\#(tr \downarrow \{read_{C1}\}) == \#(tr \downarrow \{readresp_{C2}\})$ is an example of a specification that the number of read messages must be equal to the number of readresp messages that are observed. tr represents a particular trace, # tr ↓ c denotes the number of occurrences in tr of c.

The following section illustrates these concepts with an example.

6 A SYSTEM OF SYSTEMS EXAMPLE

Let us consider a weather tracking system, M, which stores the temperature of various cities in a state. C is the system at a particular city. It reads the temperature of all the other cities every hour from M and shows it to the operator. It also measures the local temperature and sends the new temperature to M. When M receives a request for the temperature reading; it sends the details to the requester, C in this case. When it gets a new temperature to store, it notifies the notifier system B so that all other client systems (including C) can be informed about the update.

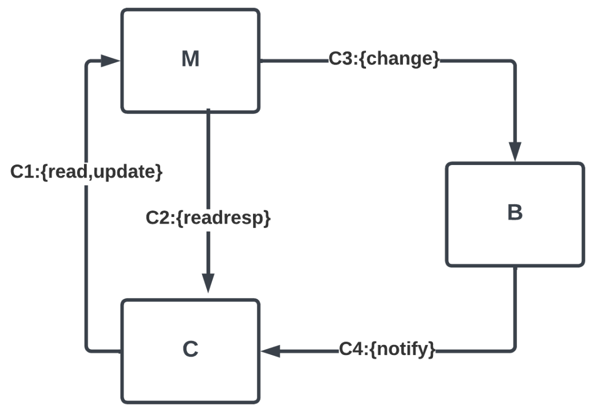

Figure 2: SoS Example

We model this as an SoS, as shown in Figure 2. M manages all the data. C sends read or update requests to M. M directly responds to the read request. M fulfills update requests and then informs B about the update so that it can further notify C (and other client systems).

6.1 Set of Systems (G)

$G = \{M, C, B\}$, three component systems are part of this SoS where

- M - The data system
- C - The system that reads and updates the data system
- B - The notifier system for informing about changes to the data system

6.1.1 *System M.* $\{X_M, X_M^0, P_M, f_M\}$

States

The state supports the following methods:

- processRead(m) - Performs the read operation and updates the state variables so that a subsequent read response message can be sent.
- processUpdate(m) - Performs the update operation and updates the state variables so that a subsequent change message can be sent.

Ports

- $MP1 = (M, in, \{read, update\})$
- $MP2 = (M, out, \{change\})$
- $MP3 = (M, out, \{readresp\})$

Transition function

```
f(x, m) =
    if (m is read) then
        processRead(m)
        send(readresp, MP3)
    if (m is update) then
        processUpdate(m)
        send(change, MP2)
```

6.1.2 *System C.* $\{X_C, X_C^0, P_C, f_C\}$

States

The state supports the following methods:

- processReadresponse(m) - Processes the read response and updates the state.
- processNotify(m) - Processes the notify message and updates the state.

Ports

- $CP1 = (C, in, \{readresp\})$
- $CP2 = (C, in, \{notify\})$
- $CP3 = (C, out, \{read, update\})$

Transition function

```
f(x, m) =
    if (m is readresp) then
        processReadresponse(m)
    if (m is notify) then
        processNotify(m)
```

6.1.3 *System B.* $\{X_B, X_B^0, P_B, f_B\}$

States

The state supports the following methods:

- processChange(m) - Processes the change message and updates the state so that a subsequent notify can be sent.

Ports

- $BP1 = (B, in, \{change\})$
- $BP2 = (B, out, \{notify\})$

Transition function

```
f(x, m) =
    if (m is change) then
        processChange(m)
        send(notify, BP2)
```

6.2 Set of Channels (L)

There are four channels as part of this SoS:

- $C1 \equiv (CP3, MP1) \equiv (C, M, \{read, update\})$
- $C2 \equiv (MP3, CP1) \equiv (M, C, \{readresp\})$
- $C3 \equiv (MP2, BP1) \equiv (M, B, \{change\})$
- $C4 \equiv (BP2, CP2) \equiv (B, C, \{notify\})$

6.3 Expected Behavior of SoS (B)

$B = T \cup R$

6.3.1 *Traces (T).* There are two important traces of this SoS:

- $READTRACE \equiv < read_{C1}, readresp_{C2} >$
- $UPDATETRACE \equiv < update_{C1}, change_{C3}, notify_{C4} >$

READTRACE captures the read scenario: C sends a read request and M responds with a readresp. *UPDATETRACE* captures the update scenario: C sends an update request, M sends a change request to B, and B sends a notify to C.

6.3.2 *Requirements and Constraints (R).* Two specifications need to be honored by all the traces of the SoS:

- $READSPEC \equiv \#(tr \downarrow \{read_{C1}\}) == \#(tr \downarrow \{readresp_{C2}\})$
- $UPDATESPEC \equiv \#(tr \downarrow \{update_{C1}\}) == \#(tr \downarrow \{notify_{C4}\})$

tr represents a particular trace, # tr $\downarrow$ c denotes the number of occurrences in tr of c [24].

READSPEC specifies that the count of read and readresp should match. *UPDATESPEC* specifies that counts of update and notify match. Notice that other constraints can also be specified: count of change and notify must match too; it is up to the modeler to choose the traces they consider for inclusion. Also, notice that if there are other ways in which System M can be updated (which will trigger a change message to B and a notify from B), *UPDATESPEC* will be violated. In other words, *UPDATESPEC* poses the requirement that System M can only be updated by System C.

By making the traces, constraints, and requirements explicit like this, we make the dynamics clearer and more comprehensible.

7 SOS MODEL OF MVC

To model MVC as SoS, we start by thinking of MVC as a transition system [13], which the user interacts with, see Figure 3. We can model user interaction as u (action) and y (observable); the user provides their inputs through u and receives the updated view through y. This model supports UC1 and UC2, as described in the previous section.

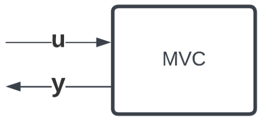

Figure 3: MVC as a system. u models the action while y models the observable for user interactions.

We decompose this top-level system into a System of Systems (SoS) comprising four component systems. In addition to View, Controller, and Model, we use an Observer system for notifications. Fig 4 shows the system model of the MVC pattern in terms of systems. The behavior of View and Controller is the same as described in Section 3.1. Instead of sending notifications directly, the Model sends an update message to the Observer system, which notifies the View system.

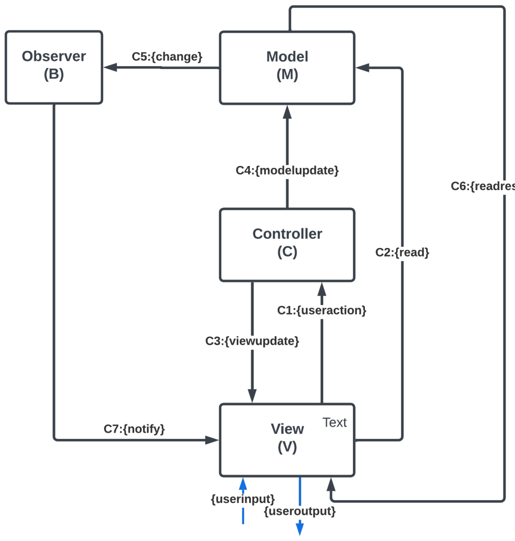

Figure 4: MVC SoS [6]. The userinput and useroutput message types are the ports for user interactions from Figure 3. These ports are not part of any channel in MVC SoS and are handled directly by View.

We model SoS as the tuple (G, L, B); following sections detail out G, L, and B.

7.1 Set of systems (G)

Four systems are part of this SoS.

$G = \{V, C, M, B\}$

- V - The View system renders the UI by fetching data from the Model and forwarding user action to the Controller. The ports with userinput and useroutput message types are for user interactions, while the messaging ports connect to other components of the MVC system.
- C - The Controller system processes all user actions and sends messages to the Model or the View for further handling.
- M - The Model system manages all access to data and responds to read and update requests.
- B - The Observer system notifies the View about changes in the Model.

Section 7.4 provides the details of each of these systems.

7.2 Set of channels (L)

There are seven channels in the SoS.

- $C1 \equiv (VP4, CP1) \equiv (V, C, \{useraction\})$
- $C2 \equiv (VP5, MP2) \equiv (V, M, \{read\})$
- $C3 \equiv (CP3, VP1) \equiv (C, V, \{viewupdate\})$
- $C4 \equiv (CP2, MP1) \equiv (C, M, \{modelupdate\})$
- $C5 \equiv (MP3, BP1) \equiv (M, B, \{change\})$
- $C6 \equiv (MP4, VP3) \equiv (M, V, \{readresponse\})$
- $C7 \equiv (BP2, VP2) \equiv (B, V, \{notify\})$

7.3 Expected Behavior of SoS (B)

$B = T \cup R$ where T captures the traces that describe the behavior, and R captures requirements and constraints on the behavior.

7.3.1 *Traces (T).*

There are two important traces of the system:

(1) $MODELUPDATETRACE \equiv < useraction_{C1}, modelupdate_{C4}, change_{C5}, notify_{C7}, read_{C2}, readresponse_{C6} >$
(2) $VIEWUPDATETRACE \equiv < useraction_{C1}, viewupdate_{C3}, read_{C2}, readresponse_{C6} >$

MODELUPDATETRACE captures a user action that results in an update to the Model: useraction message to Controller on C1, followed by modelupdate message to Model on C4, change message to Observer, notify message to View, read the message to Model, and finally readresponse message to View on C6.
Similarly, *VIEWUPDATETRACE* captures a user action that results in an update to the View.

7.3.2 *Requirements and Constraints (R).*

There are three important specifications:

(1) $USERACTIONSPEC \equiv \#(tr \downarrow \{useraction_{C1}\}) == \#(tr \downarrow \{modelupdate_{C4}, viewupdate_{C3}\})$
(2) $MODELUPDATESPEC \equiv \#(tr \downarrow \{modelupdate_{C4}\}) == \#(tr \downarrow \{change_{C5}\})$
(3) $READTRACE \equiv \#(tr \downarrow \{read_{C2}\}) == \#(tr \downarrow \{viewupdate_{C3}, notify_{C7}\})$

tr represents a particular trace, # tr ↓ {c1,c2} denotes the number of occurrences in tr of any element of {c1,c2} [24].

USERACTIONSPEC specifies that every user action must result in either a Model update or a View update. *MODELUPDATESPEC* specifies that every Model update request must result in a change notification. *READSPEC* specifies that every read request must result from either a viewupdate received or a notification received.

7.4 Component Systems description

7.4.1 *Controller (C).*

C = $\{X_C, X_C^0, P_C, f_C\}$

States

The state supports the following method:

- processUseraction(m) processes the user action and updates the state depending on whether it is a viewaction or a modelaction.

Ports

- $CP1 = (C, in, \{useraction\})$
- $CP2 = (C, out, \{modelupdate\})$
- $CP3 = (C, out, \{viewupdate\})$

Transition function

```
f(x, m) =
    if (m is useraction) then
        if (processUseraction(m) == modelupdate) then
            send(modelupdate, CP2)
        else
            send(viewupdate, CP3)
```

7.4.2 *Model (M).* M = $\{X_M, X_M^0, P_M, f_M\}$

States

The state supports the following methods:

- processRead(m) processes the incoming read request by reading from the data source. It also updates the state so that a subsequent readresponse can be sent.
- processUpdate(m) processes the incoming update request by updating the data source. It also updates the state so that a subsequent change message can be sent.

Ports

- $MP1 = (M, in, \{modelupdate\})$
- $MP2 = (M, in, \{read\})$
- $MP3 = (M, out, \{change\})$
- $MP4 = (M, out, \{readresponse\})$

System Transition function

```
f(x, m) =
    if (m is read) then
        processRead(m)
        send(readresp, MP4)
    if (m is modelupdate) then
        processUpdate(m)
        send(change, MP3)
```

7.4.3 *View (V).* V = $\{X_V, X_V^0, P_V, f_V\}$

States

The state supports the following methods:

- processReadresponse(m) processes a read response and updates the state. This will possibly impact the rendering of the view for the user.
- updateViewstate() updates the relevant state values that represent what the user sees displayed on their system; we call it displaystate.
- renderView() renders the view for the user based on the current displaystate.
- processNotify(m) processes a notify message and updates the state so that a read request can be sent to the model to fetch the details of this model update subsequently.
- processViewupdate(m) processes a viewupdate request and updates the state so that a subsequent read request can be sent to the model for the updated view.
- processUserinput(m) processes a userinput message from outside the system and updates the state so that a useraction request can be sent to the controller.

Ports

- $VP1 = (V, in, \{viewupdate\})$
- $VP2 = (V, in, \{notify\})$
- $VP3 = (V, in, \{readresponse\})$
- $VP4 = (V, out, \{useraction\})$
- $VP5 = (V, out, \{read\}$
- $VP6 = (V, in, \{userinput\}$
- $VP7 = (V, out, \{useroutput\}$

Transition function

```
f(x, m) =
    if (m is viewupdate) then
        processViewupdate(m)
        send(read, VP5)
    if (m is notify) then
        processNotify(m)
        send(read, VP5)
    if (m is readresponse) then
        processReadresponse(m)
        updateViewstate()
        send(useroutput,VP7)
    if (m is userinput) then
        processUserinput(m)
        send(useraction, VP4)
```

7.4.4 *Observer (O).* B = $\{X_B, X_B^0, P_B, f_B\}$

States

The state supports the following method:

- processChange(m) - Processes the change message and updates the state so that a subsequent notify can be sent to the View system.

Ports

- $BP1 = (B, in, \{change\})$
- $BP2 = (B, out, \{notify\})$

Transition function

```
f(x, m) =
    if (m is change) then
        processChange(m)
        send(notify, BP2)
```

8 ANALYSIS AND RESULTS

8.1 RQ1: Pattern description benchmark

"*Which source of documentation of the MVC pattern description can be considered a benchmark?*"

Based on the data in Table 1 (Section 4), POSA [11] stands out as the one having the best documentation of the MVC pattern - it provides a reasonable amount of details on the elements and their behavior. It uses a UML sequence diagram to capture the dynamics of the pattern.

We conclude that POSA [11] is the benchmark for MVC pattern description.

8.2 RQ2: Quality of the benchmark

"*Is the benchmark high-quality and fit for the needs of novice engineers?*"

We want to evaluate the benchmark pattern description in POSA [11] against the evaluation criteria laid out in Section 2.1 Q1 to Q4, and F1 to F3. Table 2 summarizes the evaluation.

Even though POSA [11] provides a good description, reliance on C++ for many of the behavioral details of the pattern hinders understanding and can pose a challenge for novice engineers. Some details of the dynamics are not readily apparent from the prose or UML diagram and are embedded in the code. The descriptions do not use a higher level of abstraction. Some requirements and constraints are mentioned in prose in some cases but not made explicit. Overall, we conclude that this is not a high-quality and fit description for the needs of a novice engineer.

8.3 RQ3: Extending the benchmark

"*How can we extend the benchmark to improve its quality and fitness?*"

Table 8.2 shows the gaps (third column) in the benchmark MVC pattern description and how the SoS model of the MVC pattern presented in Section 7 addresses these gaps (last column). Based on this, we conclude that augmenting the pattern description with an SoS model and its description makes it high-quality and fit for the needs of novice engineers. We will validate this through expert study in the future.

8.4 Other benefits of the SoS model

An SoS model offers many details about the pattern itself, which helps the teaching-learning process. Some of these are listed below:

(1) **Easy to demonstrate completeness and correctness**: Completeness can be shown by mapping the use cases (UC1 and UC2 in Section 3) with the traces specified in the Behavior (B) and correctness can be demonstrated by showing that the trace simulates the user requirement.
(2) **Easy to identify constraints and limitations**: System definitions, traces, and specifications make it easy to identify the model's limitations and, hence, the pattern. For example, *MODELUPDATESPEC* assumes that all change messages are results of modelupdate messages from the Controller; this precludes the possibility that other Controller systems work with the same model.
(3) **Easy to identify changes required to fit a need**: Given that the elements, connections, and their behavior are specified clearly, it is easy to identify changes that are needed to fit a requirement. For example, if we want the pattern to behave such that changes to the model are also notified to the Controller, we can easily see that it is not supported.

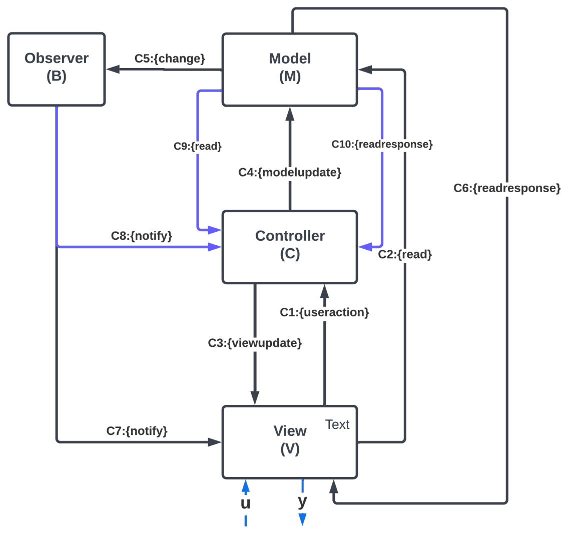

Figure 5: MVC SoS with Controller receiving model updates (blue lines show connections added to support this scenario)

9 CONCLUSION AND FUTURE WORK

Knowledge and understanding of architecture patterns can be vital to making novice engineers effective when they try to comprehend large software products. In this paper, we have proposed a way to model an architecture pattern as a system and use this model to augment the pattern documentation. Even though the approach can be used for many architectural styles and patterns, we have focused on the MVC pattern for our study in this paper. We reviewed different sources of pattern documentation and identified "Pattern-oriented Software Architecture (POSA)" [11] as the benchmark MVC pattern description.

We have defined what we mean by high-quality and fit for need in the context of a pattern description and have created evaluation criteria using the literature on pattern description. We evaluated the benchmark MVC pattern description using the evaluation criteria and concluded that it is not high-quality and fit for the needs of novice engineers. We then extended the description using a System of Systems (SoS) model of the pattern and showed that this augmented description is high-quality and fit.

Table 2: Evaluating MVC Pattern behavior descriptions in POSA vs SoS Model.

Criteria	POSA	POSA Gaps based on rubric	How the SoS Model bridges the gap
Q1: Elements of the pattern	Three components (Model, View, Controller) are listed and described.	Observer is not included in the list and is only mentioned in the code, which makes it hard to decipher.	Four component systems are listed (View, Model, Controller, Observer) and included as part of SoS definition
Q2: Connections among elements	Connections are mentioned in terms of collaborators in the class diagram,	Connection information within the class diagram doesn't make it apparent. Attributes of connection are not elaborated either.	Connections are shown using a connection diagram with channels, also explicitly included as part of the SoS definition
Q3: Behavior of the individual elements	The behaviors of individual elements are described in the implementation section, but the design decisions are not apparent.	The behavior of individual elements is not specified clearly, so there is a chance that someone misses a particular behavior.	Each system is defined as a communicating system and uses a six-tuple format. The transition function of each element captures the dynamics and behavior of the element
Q4: Behavior of the overall system	Two behaviors are identified: 1) user input resulting in model update and 2) initialization of the three components. Two constraints are indirectly mentioned, embedded in prose: 1) There can be multiple views for a model, and 2) There is one Controller for every model.	The constraints and requirements should have been captured more clearly.	The behavior of the overall system is composed of the behavior of each element and the messages flowing through the connections, which are captured as the Behavior of SoS using Traces (CSP), which makes it clear and unambiguous.
F1: Level of abstraction	Details like class objects, their initialization, and behavior in code are predominant, with minimal reference to higher-level components and systems to describe the behavior.	Higher level constructs (which are easier to follow), like components and systems messaging, are not used.	The model is described at a high level of abstraction using components and messages
F2: Modes of documentation	Most of the description is prose and C++ code	Could have used mathematical notation or pseudocode to describe the behavior; C++ implementation details become hard to follow.	The modes of documentation are predominantly set-theoretic, precise pseudocode and occasionally a specification language [24]
F3: Vocabulary	Prose and C++	While prose is easy to learn, it is not precise. C++, or any programming language, has an extensive vocabulary.	The vocabulary is small (the six tuples of communicating systems, three tuples of System of Systems, a subset of Hoare's Specification language [24])

To build the System of Systems (SoS) models, we have used transition systems vocabulary and defined an SoS model in terms of a set of Systems, Connections, and Behavior, $SoS = (G, L, B)$. We have also shown that such a model provides many other benefits.

We propose that such models be built for other architecture patterns relevant to SaaS products so that we can make all these architecture descriptions high quality and fit for novice engineers. This will go a long way in making novice engineers effective in their comprehension tasks.

Limitations and threats to validity The study is primarily done by the first author defining the evaluation criteria and applying it to the available descriptions and the benchmark (POSA) description; multiple evaluators have not been used. This evaluation could also benefit from a user study where we get users to evaluate or use the two descriptions and get feedback. Even though we looked for MVC pattern descriptions, we found very few, which resulted in a small list that we reviewed for identifying the benchmark, and we may have missed some better descriptions.

Future work. We plan to study the efficacy of the POSA and the SoS model for novice engineers and industry experts using the rubric created in this paper. We also plan to explore other relevant architecture patterns for SaaS products and build similar models. Our goal is to publish a model-augmented pattern catalog for SaaS products. We expect such a catalog to be useful for novice engineers and a good resource in the teaching-learning process of architecture patterns.

REFERENCES

[1] [n. d.]. Java SE Application Design With MVC. https://www.oracle.com/technical-resources/articles/javase/mvc.html

[2] Kevin MacG. Adams, Patrick T. Hester, Joseph M. Bradley, Thomas J. Meyers, and Charles B. Keating. 2014. Systems Theory as the Foundation for Understanding Systems: SYSTEMS THEORY AS THE FOUNDATION FOR UNDERSTANDING SYSTEMS. *Systems Engineering* 17, 1 (March 2014), 112–123. https://doi.org/10.1002/sys.21255

[3] ardalis. 2022. Overview of ASP.NET Core MVC. https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-7.0

[4] Paris Avgeriou and Uwe Zdun. 2005. Architectural Patterns Revisited - A Pattern Language., Vol. 81. 431–470.

[5] Felix Bachmann, Len Bass, Paul Clements, David Garlan, and James Ivers. 2002. *Documenting Software Architecture: Documenting Behavior:.* Technical Report. Defense Technical Information Center, Fort Belvoir, VA. https://doi.org/10.21236/ADA399792

[6] Len Bass, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice.* Addison-Wesley Professional.

[7] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. 2017. *Formal Methods for Discrete-Time Dynamical Systems.* Studies in Systems, Decision and Control, Vol. 89. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-50763-7

[8] Egon Börger. 2010. The Abstract State Machines Method for High-Level System Design and Analysis. In *Formal Methods: State of the Art and New Directions*, Paul Boca, Jonathan P. Bowen, and Jawed Siddiqi (Eds.). Springer London, London, 79–116. https://doi.org/10.1007/978-1-84882-736-3_3

[9] Ruven Brooks. 1978. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on Software engineering.* 196–201.

[10] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley Publishing.

[12] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, Chichester, UK. https://www.safaribooksonline.com/library/view/pattern-oriented-software-architecture/9781118725269/

[13] Venkatesh Choppella, Kasturi Viswanath, and Mrityunjay Kumar. 2021. Algodynamics: Algorithms as systems. In *2021 IEEE Frontiers in Education Conference (FIE).* IEEE, 1–9.

[14] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. 2002. A Practical Method for Documenting Software Architectures. (Sept. 2002).

[15] Edsger W Dijkstra. 2006. Ewd472: Guarded commands, non-determinacy and formal. derivation of programs.

[16] Christoph Fehling, Frank Leymann, Ralph Mietzner, and Walter Schupeck. 2011. A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-Based Application Architectures. *Institute of Architecture of Application Systems* (2011).

[17] Martin Fowler. 2012. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch.* Addison-Wesley.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7.* Springer, 406–431.

[19] Yuri Gurevich. 2000. Sequential Abstract-State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic* 1, 1 (July 2000), 77–111. https://doi.org/10.1145/343369.343384

[20] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (June 1987), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

[21] Cindy E. Hmelo-Silver and Roger Azevedo. 2006. Understanding Complex Systems: Some Core Challenges. *Journal of the Learning Sciences* 15, 1 (Jan. 2006), 53–61. https://doi.org/10.1207/s15327809jls1501_7

[22] Cindy E. Hmelo-Silver, Surabhi Marathe, and Lei Liu. 2007. Fish Swim, Rocks Sit, and Lungs Breathe: Expert-Novice Understanding of Complex Systems. *Journal of the Learning Sciences* 16, 3 (June 2007), 307–331. https://doi.org/10.1080/10508400701413401

[23] Cindy E. Hmelo-Silver and Merav Green Pfeffer. 2004. Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions. *Cognitive Science* 28, 1 (Jan. 2004), 127–138. https://doi.org/10.1207/s15516709cog2801_7

[24] Charles Antony Richard Hoare et al. 1985. *Communicating sequential processes.* Vol. 178. Prentice-hall Englewood Cliffs.

[25] Jon Holt and Simon Perry. 2008. *SysML for Systems Engineering.* Vol. 7. IET.

[26] Ahmad Waqas Kamal and Paris Avgeriou. 2007. An Evaluation of ADLs on Modeling Patterns for Software Architecture. In *EPRINTS-BOOK-TITLE.* University of Groningen, Johann Bernoulli Institute for Mathematics and ....

[27] Glenn E Krasner. 1988. A cookbook for using model-view-controller user interface paradigmin smalltalk-80. *J. Object Oriented Programming* 1, 3 (1988), 26–49.

[28] Mrityunjay Kumar and Venkatesh Choppella. 2023. Evaluating the difficulty for novice engineers in learning and using Transition Systems for modeling software systems. In *Proceedings of the 16th Annual ACM India Compute Conference.* 19–24.

[29] Mrityunjay Kumar and Venkatesh Choppella. 2023. A modeling language for novice engineers to design well at SaaS product companies. In *Proceedings of the 16th Innovations in Software Engineering Conference.* 1–5.

[30] Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and Software* 7, 4 (Dec. 1987), 325–339. https://doi.org/10.1016/0164-1212(87)90032-X

[31] Nancy Lynch. 2003. Input/Output automata: Basic, timed, hybrid, probabilistic, dynamic,.... In *CONCUR 2003-Concurrency Theory: 14th International Conference, Marseille, France, September 3-5, 2003. Proceedings 14.* Springer, 191–192.

[32] Nancy A Lynch. 1996. *Distributed algorithms.* Elsevier.

[33] Ian MacColl and David Carrington. 1999. Specifying interactive systems in Object-Z and CSP. In *IFM'99: Proceedings of the 1st International Conference on Integrated Formal Methods, York, 28-29 June 1999.* Springer, 335–352.

[34] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2013. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering* 39, 6 (June 2013), 869–891. https://doi.org/10.1109/TSE.2012.74

[35] Donella H Meadows. 2008. *Thinking in systems: A primer.* chelsea green publishing.

[36] Robin Milner. 1980. *A calculus of communicating systems.* Springer.

[37] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. 2018. Architectural Principles for Cloud Software. *ACM Transactions on Internet Technology (TOIT)* 18, 2 (2018), 1–23.

[38] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4 (Oct. 1992), 40–52. https://doi.org/10.1145/141874.141884

[39] Mary Shaw and David Garlan. 2005. Formulations and formalisms in software architecture. *Computer Science Today: Recent Trends and Developments* (2005), 307–323.

[40] Graeme Smith. 2012. *The Object-Z specification language.* Vol. 1. Springer Science & Business Media.

[41] J Michael Spivey and Jean-Raymond Abrial. 1992. *The Z notation.* Vol. 29. Prentice Hall Hemel Hempstead.

[42] Paulo Tabuada. 2009. *Verification and control of hybrid systems: a symbolic approach.* Springer Science & Business Media.

[43] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2018. Architectural Patterns for Microservices: A Systematic Mapping Study. In *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018.* SciTePress.

[44] Johan Van Benthem and Jan Bergstra. 1994. Logic of Transition Systems. *Journal of Logic, Language and Information* 3, 4 (Dec. 1994), 247–283. https://doi.org/10.1007/BF01160018

[45] Tim Weilkiens. 2011. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* Elsevier.

[46] Dharmendra K Yadav and Rushikesh K Joshi. 2010. Capturing interactions in architectural patterns. In *2010 IEEE 2nd International Advance Computing Conference (IACC).* IEEE, 443–448.

[47] Bernard P Zeigler, Tag Gon Kim, and Herbert Praehofer. 2000. *Theory of modeling and simulation.* Academic press.